The Oxbridge Formula

Oxbridge Computer Science Interview Scripts

What's the hardest thing about preparing for an Oxbridge interview?

More than even the difficulty of the questions themselves, many students say it's the unfamiliarity of the whole experience.

How can you prepare if you'll have no idea what it will be like?

These example interview scripts are a part of our solution to this problem. We consulted with multiple Oxbridge graduates and current Oxbridge students to create examples of how a good student might respond to a typical interview-style question, and how the tutor would both test and support them throughout.

We've also added further information on what each question is testing, how it might be extended, how the tutor might give further hints if the candidate got stuck at any point, and what the student does well that you might look to replicate in your own interview.

To get the most out of these scripts, we advise you to think about how you'd respond to the question, and to each of the tutor's hints, comments, or further questions, before carefully reading the student's response, and then making sure you understand what's going on before you move on to the next part.

Of course, while reading a script of an interview can be very useful, it's not the same as having to answer the questions yourself in real time!

That's why we're also running live online mock interview courses and 1-1 sessions for Computer Science this November and December.

Introduction

Below are three example Computer Science interview questions worked through by a model student. If you want to practise answering interview-style questions, try to answer the question as best you can on your own before working through the script, and when you do read through, try to do each step before you read the student's response. Remember, with interview questions, you aren't expected to be able to do them all right away—the interviewer wants to be able to talk to you about the questions and work them through with you.

Computer Science interview question 1

(Recursion/Iteration)

Question: The 'iterated logarithm', log^*, is defined as the number of times one needs to apply the base-2 logarithm to a number before it is less than, or equal to, 1. I'd like you to find a recursive formula for log^* and use this to find $log^*(8)$.

Many Computer Science interviews focus on mathematical questions, rather than anything explicitly linked to programming or technology.

Model interview

Tutor: Are you familiar with the definition of recursion?

Don't panic if you're not! The interviewer will explain unfamiliar terms—they're testing what you can learn, not what you already know.

Student: Yes, it's when something is defined in terms of itself.

This might not be the most precise answer ever, but it sounds relevant enough that the interviewer can wait to see what the student comes up with to see if they really do understand.

Tutor: Okay, so let's find a recursive formula for log^*.

Student: Recursive formulae have base cases. In this case, we stop applying log when the number is less than 1. So, we have $log^*(n) = 1$ if $n < 1$.

Tutor: Is that correct?

Student: (*Pauses.*) Oh, should it be 0 instead of 1? (*Thinks.*) Yes, it should be, because you don't need to apply log any times once the number is less than 1... or equal to 1. So it should be $log^*(n) = 0$ if $n \leq 1$.

Tutor: Okay, and otherwise?

Student: Well, that's the base case. If we have a number bigger than 1, then we apply the logarithm. And then we want to find the iterated logarithm of that number. So we have $log^*(log\ n)$... And then... (*Pauses.*) I'm not sure.

Tutor: If I gave you the value of $log^*(log\ n)$, call it k, what is the value of $log^*(n)$?

Student: Well, log applied k times to $log(n)$ is less than or equal to 1... and then log applied $k + 1$ times to n is less than or equal to 1. So it's $log^*(n) = log^*(log\ n) + 1$. And that applies for $n > 1$.

Tutor: Okay, good. So then what is the iterated logarithm of 8, using this formula?

Student: So if we start with 8 and take the logarithm... and we're using base 2... then $2^3 = 8$ so we get 3. Applying the logarithm again... *(Pauses.)* We're looking for a number where $2^x = 3$. But this isn't going to be a whole number.

Tutor: Right. Can we work out an upper bound and a lower bound for x?

This is often a useful technique in integer questions.

Student: *(Pauses to think.)* $2^1 = 2$ and $2^2 = 4$ so it should be between 1 and 2.

Tutor: Yes. What's the next step?

Student: If we take the logarithm of 1 we get 0, which is less than 1.

Tutor: Does the number need to be less than 1?

Student: *(Looks at formula again.)* Oh no, actually it can be equal to 1, so we stop there.

Tutor: And that means the iterated logarithm of 1 is?

Student: 0?

Tutor: That's right. And the iterated logarithm of 2?

Student: It's $log_2\ 2 = 1$, and we stop at 1, so it's 1?

Tutor: Yes. What about the iterated logarithm of 8?

Student: We do it once to get 3, and a second time to get a number between 1 and 2 and a third time to get a number between 0 and 1 so it's between 2 and 3.

Tutor: Can the iterated logarithm be a non-integer?

Student: Oh, it's the number of times you apply the logarithm so it has to be a whole number. So with 8 we either did it 2 or 3 times.

Tutor: Which one, 2 or 3?

Student: Well it depends if it's 1 or 2 after we've done it twice.

Tutor: Can it be exactly 1 or 2?

Student: *(Thinks for a second.)* No, because it's bigger than 2 and less than 4.

Tutor: Yes, so we know that our x satisfying $2^x = 3$ is more than 1 and less than 2.

Student: And we do have to apply the iterated logarithm a third time, so it's 3.

Interviewers will often ask questions that generalise the results you've worked out.

Tutor: Yes. Now please find all numbers that have an iterated logarithm of 3.

Student: Okay. So those are the numbers that have

$loglog\log n \leq 1$. So if we raise each side to the power of 2, we get $loglog\, n \leq 2$ and then $log\, n \leq 4$ and then $n \leq 16$.

Tutor: Is every number less than or equal to 16 suitable?

Student: No, because some might have an iterated logarithm of 0, 1 or 2. So we also need $loglog\, n > 1$. That's the same as $log\, n > 2$ or $n > 4$. So n has an iterated logarithm of exactly 3 if it's in the range $4 < n \leq 16$.

Tutor: Okay, thank you.

Further hints

- The interviewer might prompt you to give an example of a recursive formula. If you're stuck at first, you can establish what properties recursive formulae have (they consist of base cases and recursive cases).
- If you're stuck on how to find all numbers with iterated logarithm 3, the interviewer might ask you something like: 'What is the smallest number with iterated logarithm 3?'

Extending the question

When you take the equation $loglog\log n \leq 1$ and raise each side to the power of 2 to get $loglog\, n \leq 2$, this only works because 2^x is an 'increasing function'. This means that if $x > y$, then $2^x > 2^y$. Not every function is increasing (e.g. $f(x) = -x$ and $f(x) = cos(x)$ are not), so you can't just apply any function you want to both sides of an inequality. The interviewer could ask follow-up questions about this if it comes up in the discussion (for instance, if you ask if you're allowed to raise each side of an inequality to the power of 2).

In fact, log^* itself is an increasing function. This lends itself to another way of deriving the range in which numbers have an iterated logarithm of 3: find the smallest numbers with iterated logarithms 3 and 4, and any number with iterated logarithm 3 is between these values. Calculation shows that $2^2 = 4$ and $2^{2^2} = 16$ are these bounds, so we get the interval $4 < n \leq 16$ as before.

If there's time, the interviewer could ask a follow-up question: 'For a positive number k, find all numbers n such that $log^* n = k$.' The solution (which the interviewer will help the student construct) is to consider a 'power tower' of k lots of 2s stacked on top of each other (e.g. for $k = 4$, we have $2^{2^{2^2}} = 2^{16}$ [=65,536] as the upper bound). Write this number as $Tower\ (2, k)$. The numbers n that have iterated logarithm k are the ones satisfying $Tower\ (2, k-1) < n \leq Tower\ (2, k)$.

What is the question testing?

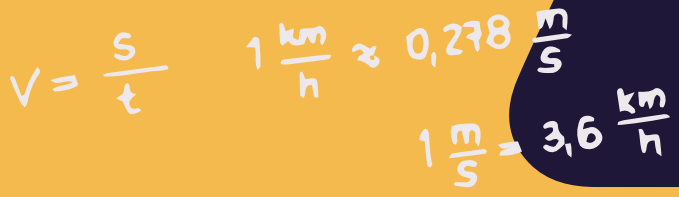
The question tests your ability to learn a new concept and apply it to your existing knowledge of logarithms. It also tests your understanding of recursion and your ability to derive formulae.

Related topics from university

Recursion is one of the most common techniques for writing algorithms and occurs widely throughout Computer Science as well as mathematics.

The iterated logarithm is used in Computer Science as a measure of 'complexity' of some algorithms—that is, how quickly they run. If you've come across 'big O notation', the iterated logarithm (written log^*) is interesting as it's a function that grows very, very slowly—much slower than any function in $O(log\ n)$ or even $O(loglogloglog\ n)$.

The additional material relating to 'power towers' relates to a function in arithmetic known as 'tetration'.



Computer Science interview question 2

(Graph Sketching/Inequalities)

Question: Put the following functions in increasing order in terms of how fast they grow as n tends to infinity: $100n^3$, n^4, $\log n$, $(\log n)^{\log n}$, $\log\log n$, $n^{\log n}$.

Model interview

Student: When the question says 'log', does it mean 'base 10 *log*'?

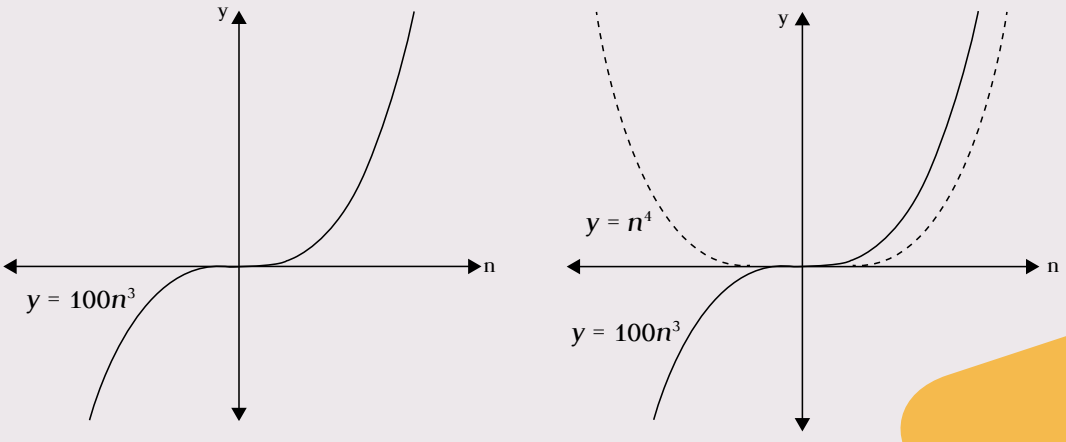
Tutor: In fact, it doesn't matter—but let's say it's the common logarithm. That is, the base 10 logarithm.

Student: Okay. (*Pause.*) I'll start by sketching the graphs of some of these functions.

Tutor: Good.

Student: So here's $y = 100n^3$. And $y = n^4$.

Graphs can just be rough sketches, unless you're asked for more detail. Remember to always label axes though!



As for which one is bigger... we're looking at the right-hand side of the graph and it looks as though $100n^3$ is the one on the inside so it's the one that's growing fastest.

Tutor: Is that right?

Student: No, actually shouldn't n^4 grow faster?

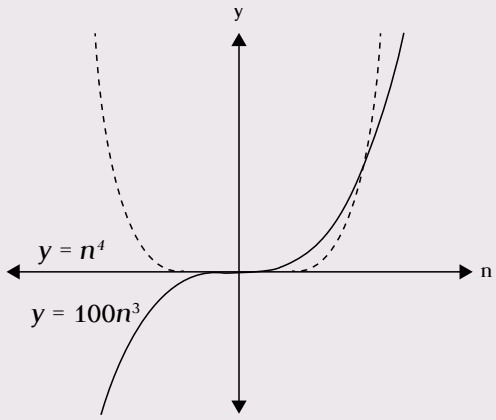
It's okay to change your answer or to be unsure. It's best to say your vocalise thoughts rather than sitting in silence so the interviewer can see how you're thinking.

Tutor: Why would that be?

Student: Because 4 is a higher power than 3. (*Pause.*) Okay, so maybe n^4 overtakes $100n^3$ higher up.

Tutor: How can you tell if that happens?

Student: Well, if we solve $100n^3 = n^4$ we can see where they intersect. And we get $n^3(n - 100) = 0$, so they intersect at 0 and 100. (*Pause.*) So the graph of n^4 overtakes $100n^3$ past 100.



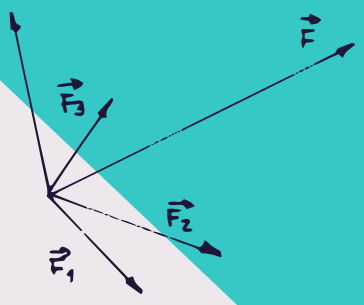
Tutor: Okay, so which is growing slower?

Student: $100n^3$.

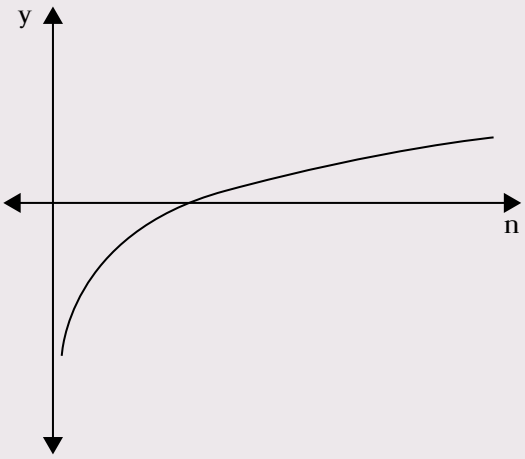
Tutor: What about the other functions?

Student: Well, $log\ n$ is going to grow very slowly and $loglog\ n$ will grow even slower.

Tutor: Why is that?



Student: If I draw $log\ n$...



It's growing slower than $100n^3$ or n^4. And if we take the logarithm of that, it becomes even smaller... (*Pause.*) because $y = log\ n$ grows slower than $y = n$, so if we replace n with $log\ n$ we get that $loglog\ n$ grows slower than $log\ n$.

Tutor: Okay. How about the remaining functions?

Student: By the same reasoning, $(log\ n)^{log\ n}$ must grow more slowly than $n^{log\ n}$. And if we compare $(log\ n)^{log\ n}$ to n^4... firstly, we see that n^4 grows slower than $n^{log\ n}$ because $log\ n$ will be bigger than 4 as n gets bigger. Finally, to solve $(log\ n)^{log\ n} = n^4$, I'll put 10 to the power of each side. That means the left-hand side is $10^{(log\ n)^{log\ n}} = 10^{(log\ n)^2}$...

Is this right? Make sure you're very confident with the rules for logs and powers!

Tutor: What if, instead, we try to take the logarithm of the equation $(log\ n)^{log\ n} = n^4$?

The interviewer may give you advice if you go down the wrong track.

Student: I think that $log((log\ n)^{log\ n}) = (log\ n)(loglog\ n)$ and $log(n^4) = 4\ log\ (n)$.

That means $(log\ n)^{log\ n} > n^4$ if $(log\ n)(loglog\ n) > 4\ log\ (n)$, which is true if $loglog\ n > 4$, which is true for $n > 10^{10^4}$. Therefore, $(log\ n)^{log\ n}$ grows faster.

Tutor: Okay, thank you.

Note: Final order (slowest to fastest) is: $loglog\ n$, $log\ n$, $100n^3$, n^4, $(log\ n)^{log\ n}$, $n^{log\ n}$.

Extending the question

The interviewer could introduce more functions such as: $(log\ n)^n$, 2^n, $\sqrt{n}$, or $(log\ n)^{(\log n)^{\log n}}$.

They could also ask you to compare the derivatives of some of the functions (treating n as a continuous variable) to test your calculus skills too.

What is the question testing?

The question requires a thorough understanding of logarithms, exponentials, polynomials and graph sketching.

Related topics from university

The question relates to 'big O notation', a way of classifying functions in terms of how fast they grow. This is important in Computer Science as a way of classifying how efficient an algorithm is. It's also used in analysis, an area of mathematics, as a way to find limits.

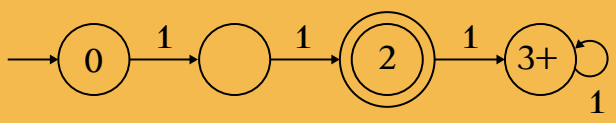
You may have already encountered Big O notation in your Computer Science A Level or elsewhere. If this is the case, the interviewer may add more functions or continue pushing to find an area you're unfamiliar with. The interviewer cares more about finding out how you think (and respond to new information) than how much information you already know.

Computer Science interview question 3

(Algorithms/Automata)

At the beginning of the interview, the interviewer hands the candidate the question on a piece of paper and asks them to read the material.

A *deterministic finite automaton* (DFA) is a machine that can be in one of a finite number of states. It starts in an initial state and changes state according to the input it receives. Some states are *accepting*: if the machine is in an accepting state when the input terminates, then it 'accepts' the input. Otherwise it 'rejects' the input.



The automaton represented by this diagram can only recognise strings of 1s. The arrow going into the state 0 indicates that 0 is the start state. The double circle around the state 2 indicates that it's an accepting state. An arrow from state X to state Y labelled with a 1 indicates that when the machine is in state X and receives a 1 as input, it moves to state Y.

For instance, with the input 111, the machine moves from state 0 to 1 to 2 to $3+$, and as $3+$ isn't an accepting state, the machine rejects 111.

Questions:

i) What strings does this machine accept?

ii) Let 1^n be the string consisting of n repetitions of the character 1, where n is a fixed number. Explain how to construct a DFA that accepts only the string 1^n.

iii) How can this be extended to make a DFA that accepts a finite set of strings: 1^a, 1^b, ..., 1^z?

Model interview

Student: It seems the only thing this machine accepts is the input 11.

Tutor: Correct, and how can we show that?

Student: If we put in 11 then the machine goes from state 0 to 1 to 2, and 2 is an accepting state, so the machine accepts 11.

In a case like this, it's important to explain each step of the algorithm or process, even if it seems easy at first, because the tutor needs to see that you're thinking clearly about how it works.

Tutor: Okay. And if we input something else?

Student: Well, if we put in three or more 1s then the machine goes to state 3 + and stays there forever. And 3 + isn't an accepting state so it will reject the input. And if we put in just the input 1 then it goes to state 1, which isn't accepting.

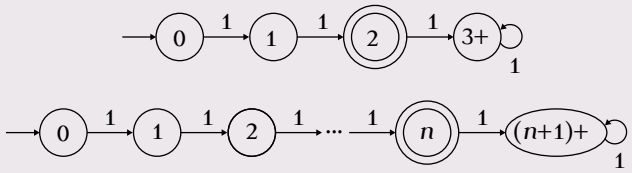
Tutor: Okay. We can also input the empty string.

Always be on the lookout for special cases like this!

Student: Meaning zero 1s? Then the machine stays where it started, in state 0. And 0 isn't an accepting state.

Tutor: Okay, good. Now I'd like you to read question ii.

Student: (*Reads the question.*) The original machine solves this problem when n is 2. And we can do the same method for any n.

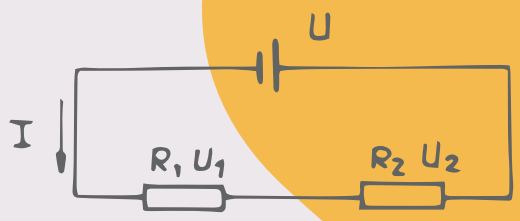


Tutor: Yes. And if we want to change this to accept the strings 1^a, 1^b, ..., 1^z?

Student: We can change the accepting state from n to the numbers a, b, ..., z.

Tutor: Okay. And what is n in this case?

Student: We could just get rid of n and have one state from each number 0, 1, 2, ...



Tutor: In that case, the machine isn't finite.

Student: Oh, okay. Then we can just go up to the largest number out of $a, b, ..., z$.

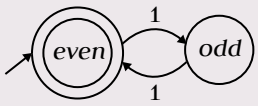
Tutor: Yes, good. Next, I'd like you to make a DFA that accepts strings of the form 1^n where n is a multiple of 2.

The tutor is now extending the question beyond what was asked on the paper the student was given at the start. Your interviewers may well keep adding to a question in order to stretch you.

Student: Right. We can't have a separate accepting state for each of 1^0, 1^2, 1^4, ... because that wouldn't be finite.

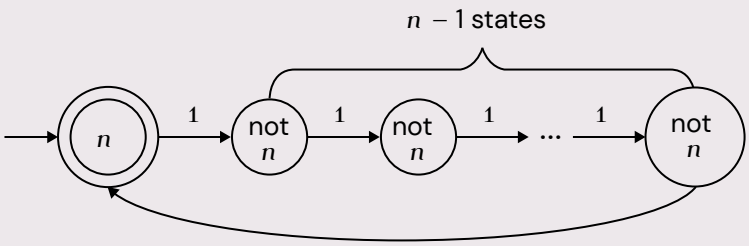
Tutor: No.

Student: (*Thinks.*) We only need to know whether the number of 1s inputted is an even or odd number. We can have one state for $even$ and one for odd. It would look something like this:



Tutor: Yes. And if instead of multiples of 2, what if we looked at multiples of n?

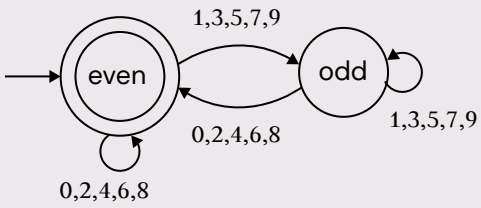
Student: Then we need n states. So, would this work?



Tutor: Yes, although it might have been better to label the states $0, 1, ..., n - 1$. Right, now I'll ask you to think about DFAs that take as input not just a sequence of 1s but a sequence of digits $0, ..., 9$. In that case, the number 981 would be represented by inputting a 9, an 8 and a 1. Our diagram can have one arrow per input digit per state. I'll ask you to make a DFA that accepts decimal numbers[59] that are a multiple of 2.

If you've ever studied modular arithmetic, this might seem familiar.

Student: Okay. Again, we just need to know whether the input is even or odd. And we can keep track of that by seeing whether the last digit is $0, 2, 4, 6, 8$ or $1, 3, 5, 7, 9$. The DFA would be:



[An arrow with '1, 3, 5, 7, 9' is shorthand for 5 arrows with 1, 3, 5, 7 and 9 as labels.]

Tutor: Good. Now how could we write a DFA to determine if a number is a multiple of 3?

Student: We can't do this just by looking at the last digit, because multiples of 3 can end in any digit. There's a test to see if a number is divisible by 3 by adding all of its digits together.

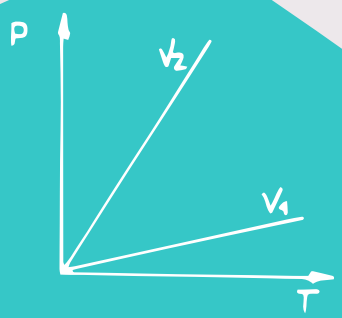
Tutor: And how do you tell if the number is divisible by 3 with that test?

Student: A number is divisible by 3 if you add all its digits together and get a multiple of 3. Or, if that number is still too big to work out if it's divisible by 3, you can keep adding its digits together until you get down to a single-digit number.

Tutor: Okay. How can we use this idea to construct a DFA?

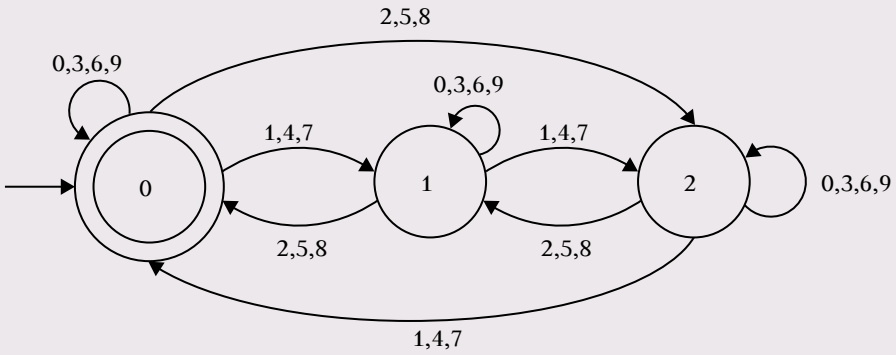
Identifying the obstacle you're facing is a good thing to do when you don't have a complete answer yet.

Student: Well, we can just keep a running count of the sum of the digits. But we can't store the number because we don't know how big it will be.



Tutor: Okay. Do we need the whole number? (*Waits.*) Or just a property of it?

Student: Oh, we only need to know whether or not the running total is a multiple of 3. Or, whether it's 1 or 2 more than a multiple of 3. If the next digit is 0 mod 3, the state doesn't change. If it's 1 mod 3, the value of the running total mod 3 increases by 1, and so on. So it would look like this.



[Note: The interviewer would not make the student add all of the labels, only those necessary to explain how the DFA works.]

Tutor: Okay, thank you.

Further hints

The interviewer will prompt you to correct any small mistakes you've made, such as forgetting to draw an arrow for the initial state, or a double circle for accepting states. If you were stuck on the last part, the interviewer might prompt you to think of any divisibility tests for 3 that you might know—and if you don't know any, the interviewer will likely explain one.

Extending the question

As well as multiples of 2 and 3, we can make a DFA to test divisibility by any number for which we have a divisibility rule involving the digits they end in, or some type of digit sum.

For instance, a number is divisible by 8 if its last three digits are divisible by 8. For example, $713{,}564{,}256$ is a multiple of 8 because 256 is a multiple of 8. We can have one state for each of the last three digits, from 000 to 999, and make all the multiples of 8 (000, 008,..., 992) accepting states. However, this does require a DFA that has $1{,}000$ states. Can you think of a DFA that does the same thing with fewer states?

A number is divisible by 11 if its alternating digit sum is a multiple of 11 (e.g. $83{,}929{,}032$ is a multiple of 11 because $8 - 3 + 9 - 2 + 9 - 0 + 3 - 2 = 22$ is a multiple of 11). How can we make a DFA for multiples of 11 using this rule?

Additionally, given two DFAs, M_1 and M_2 that accept multiples of m and n, we can make a DFA M_3 that accepts multiples of mn. Can you work out how? (Hint: if we enter a number and this causes M_1 to move to state x and M_2 to move to state y, we want M_3 to be in a state (x, y).)

What is the question testing?

You may already be familiar with deterministic finite automata, perhaps under the name 'finite state machines', but the interviewer is testing how well you can learn new information. As a result, if you already know some of the background material, you'll be pushed further into the extension material to a point where you're asked a question you haven't come across before.

The question tests your ability to learn new information quickly and to design abstract procedures. Candidates with programming knowledge may have written algorithms to test divisibility of numbers, but with DFAs there's a constraint that only a finite number of states can be used, making things more difficult.

Related topics from university

Deterministic finite automata are used in the theory of computation in Computer Science. We say a language (set of strings) is decided by a DFA if the DFA accepts every string in the language and rejects every string that isn't in the language. Not every possible language can be decided by a DFA—the languages that can be are called 'regular languages'.

At Cambridge, DFAs are covered in the Digital Electronics and Discrete Mathematics modules in the first year. At Oxford, the second year Models of Computation course covers DFAs and their properties, as well as *non*-deterministic finite automata and several other types of theoretical machine.

$$F = |\vec{F_1} + \vec{F_2}| = \sqrt{F_1^2 + F_2^2 + 2F_1F_2\cos\alpha}$$

$$\epsilon(\lambda) = \frac{2\pi c^2}{\lambda^5}\frac{h}{e^{\frac{hc}{\lambda kT}} - 1} = \frac{c_1}{\lambda^5 \cdot e^{c_2/\lambda T} - 1}$$

If you found these scripts helpful and you're looking for more support for you interview prep, why not check out our live online mock interview courses and 1-1 sessions for Maths?

Oxbridge Computer Science Interview Preparation Options

- Oxbridge Computer Science interview course
- 1-1 Oxbridge Computer Science mock interview sessions
- 1-1 tuition sessions

The Oxbridge Formula